Reverex DX

Networking Capabilities

Student Name: Colin Doubrough

Student ID: 7140350

Course: COSC 3P99

Instructor: Renata Queiroz Dividino

**Table of Contents**

[**Project Goal 2**](#_g5wrfjj9mozd)

[**Challenges: Networking challenges within Reverex. 3**](#_wbk5c4axyqad)

[**To (accidentally) create a Toolkit. 3**](#_wzpnea3keaq5)

[Synchronized Variables. 3](#_oadwpnjmntp5)

[Problems. 4](#_1p7gdhwrtrk3)

[Learn your RPC’s 4](#_3ianfsyp8l2g)

[NetAction 5](#_7ukgj6z0qhl8)

[Benefits 5](#_7npbd486jy81)

[Mistakes 5](#_b9tmx6956ba)

[If at first, you don’t succeed, abstract, abstract, abstract! 6](#_76f5xopvjlkp)

[NetRoutines 6](#_y5izt9ujeju6)

[Results. 6](#_gryvc7kxrbr1)

[Conclusions 7](#_chhop2ny8jp9)

[**Citations. 8**](#_z89rz1dt9b8s)

# **Project Goal**

The goal of this project is to enhance the existing REVEREX game with networking capabilities to enable a 2-player online multiplayer mode. Partial source code is provided.

# **Challenges: Networking challenges within Reverex.**

Developing any video game is an incredible challenge and undertaking. Only some games make it from conception to release. However, creating a Networked Game is another layer of complexity and challenge.

Initially, the team had planned for networking to take a backseat compared to just working on porting existing feature sets. However, it quickly became apparent if we didn’t start networking early, we would just create a large amount of technical debt to work on later. This was mainly due to the lack of manpower assigned to do networking. I was the only person assigned initially to do any form of networking for the project. Moreover, the networking API we decided to use was the rather low-level Steam Networking API [[1]](#_khnr4py0uj1c), which didn’t provide anything higher level than string communication and serialization.

While the package was remarkably simplistic and easy to start with, it was apparent that just using Steam’s API [[1]](#_khnr4py0uj1c) was incredibly difficult to scale with. Having to teach every single developer and onboard them with concepts like serialization, string manipulation, and conversions from Byte arrays to strings would’ve taken an eternity and would’ve likely had low participation from most of our developers.

So we needed a system that could scale well, and we needed that system done fast if we were to meet any of our programming goals set up at the beginning of the year.

# Development: (Accidentally) creating a Toolkit.

However, I did not start with a big plan to create a batch of networking tools. Instead, I started by simplifying branching statements for whether the player was the first or second player. So, I created Whoami [[A.1]](#_lb3hfxekrpxw), a class that contains several static helper methods that can ask basic questions, like whether players 1 or 2 are locally present or playing across a network and if the game is online or in a local setting.

These helper functions were necessary to make basic raw serialization not pain-inducing. However, that still left the issue that I was the only person working on networking for the entire project unaddressed, and as development moved further along a lot of stuff started to get blocked because I couldn’t be everywhere at once.

I needed to create something others could use and complete it quickly so development could speed up.

## Synchronized Variables.

What is networking at its core other than just a series of tasks that ought to be done to serialize data and send it between two machines? Ultimately, this is a task that could easily be abstracted away into smaller pieces. I’ve worked on several mods [[2]](#_s8cj2gghhjba) for games like Minecraft in my spare time, some small, some larger and some abandoned. One thing I’ve noticed is that Moddable games with networking often provide classes that could do that data serialization and sending/receiving for you.

Knowing this, I started looking into the robust serialization libraries offered by Unity [[3]](#_9b4h97aox49z)[[4]](#_kpqh1x8v88wf). Since Unity’s libraries supported serializing several common types, I could just create a class that can generically use Unity’s libraries to handle serialization and deserialization. Afterwards, I would just need to write code to differentiate between different object instances and figure out where on the other machine to store that incoming data.

Ultimately, there were two approaches I took, with the first one being too naive to work.

I started with the chronological association. I initially tried to differentiate objects depending on when they are inline created, so if an object is the 50th object created with our toolset, it would have the identifier 50. A chronological approach was especially easy, and while it does work when dealing with one class as they’ll create inlined objects from the first line down, once you start creating Synchronized Variables outside of the scope of just one class it becomes possible for two machines to have different execution orders even while running the same program.

Afterwards, I decided to be less fancy and force programmers to assign synchronized [[A.2]](#_w8u3amuzbmbm) variables with English-readable names. Then, I’d hash the name before adding it to the data structure containing all synchronized variables used in the program. That hash then becomes a unique identifier that I could use to associate data with the appropriate object in memory, allowing me to easily store the data coming from another machine.

### Problems.

While Synchronized [[A.2]](#_w8u3amuzbmbm) was significantly more abstract than dealing with raw data and was a pretty cool code tool to show off, it wasn’t the most practical tool. It could undoubtedly simplify relatively straightforward code tasks like storing the state of some objects. Unfortunately, it lacked the tools necessary to do more complex tasks and couldn’t call code to run on a remote machine without constant polling, which isn’t particularly performant nor optimal. Lastly, no one used the tool I provided because it was too limited, had a few nasty bugs and wasn’t convenient for most tasks that needed to be done. Unfortunately, I was likely still doing around 90% of networking tasks at this point.

## Learn your RPC’s

After constructing Synchronized [[A.2]](#_w8u3amuzbmbm), it became apparent we still needed more tools. Specifically, within our current networking package, we need a way to flag certain events to run on another or potentially both machines at the same time. Our previous system could differentiate different types of objects, but we needed to differentiate between different methods.

After realizing how relatively simple implementing Synchronized was, I started adapting that system to work with method calls rather than just objects in memory. Eventually, I came across a syntax that I thought would be relatively understandable for other developers.

### NetAction

In C#, an action is an object that contains references to other methods in memory and will call each and every one of those methods when invoked. Since the same objects should exist with the same Unique Identifiers on both machines, as long as we tell those objects to hook into the same methods on both machines, it should be possible to send a packet including something that identifies an action and use that to call methods remotely from another device.

The main concept behind NetAction [[A.3]](#_lndue54jkbht) is to write a wrapper surrounding an action to automatically handle requiring an English-readable name and the dirty aspects of serialization, deserialization and message passing. By abstracting these aspects we limit the amount of work any developer needs to do to just get data transfer working to creating an object, adding its listeners, and calling invoke.

I also added support to transfer a serializable object/struct alongside the method invocation to work as parameters for the request. If you need more parameters than one, you’d add them to the struct you’d be passing alongside the request.

### Benefits

NetAction was a significant improvement to user experience when writing net code. For the first time, programmers could write net code without always deferring to me to help them produce it. It took a massive weight off my shoulders and allowed me to focus more on getting more tasks done rather than hopelessly trying to tackle the entire backlog of networking tasks.

### Mistakes

The API was still extremely clunky and verbose, I had used quite a few complicated C# aspects like Lambdas and Actions that went over the heads of some of the developers, and as a result, few tried to learn the new API. Moreover, revolving the whole usage of this API around actions was undoubtedly a poor move, as the limited experience surrounding actions resulted in many hours of weird bugs I had to fix due to developers forgetting to unsubscribe from actions before deleting an object.

Ultimately, I was still probably doing 50% of most networking tasks, which was not ideal considering the group size.

## Learned Lessons: If at first, you don’t succeed, abstract, abstract, abstract!

At this point, I had created the entire system and wanted to further improve it to make it easier to use and understand in comparison to what it currently was. However, since many of the required systems were already in place, and impossible to remove or revamp without hindering the development timeline even further. I was struggling to figure out some way of integrating changes within our current codebase.

A suggestion that came from Renata was to create an abstraction on top of NetAction.

As a result, I decided to take an iterative approach and create a new abstraction layer for NetAction that would simplify the syntax considerably and convert the entire process of writing Netcode to something easy enough for everyone on the team to grasp.

### NetRoutines

To do this, I started by drafting out what NetRoutine [[A.4]](#_ynu50sooksn3) needed and began communicating the changes with the programming team.

Some of the goals of this abstraction were to smooth over the obtuseness of NetAction, remove the lambda’s and move ownership checks to be internal aspects of the class. Implicitly naming the underlying NetAction using reflection and finally abstracting away the Action into just a simple method invocation.

Most of these goals weren’t too hard to meet as they were simple adjustments added to the NetRoutine wrapper. Implicit naming had some difficulties since it reduced the scope of what you could do with NetAction [[A.3]](#_lndue54jkbht). In the end, I decided to make implicit naming only happen if we could assign a method to the NetRoutine [[A.4]](#_ynu50sooksn3) at compile time and allowed users to supply their name if they wanted to use them as a conventional NetAction instead of a NetRoutine.

Lastly, I made my changes to the syntax to move away from users interfacing directly with actions and instead interact with NetRoutines similarly to how Corotuines [[5]](#_fjpfjzhlg2w2) work under Unity.

### Results.

Ultimately, I was successful in creating the abstraction layer. NetRoutines created a much friendlier way of creating new NetActions and allowed more of the development team to understand how to use our Networking Tools. NetRoutines massively reduced the effort needed to work on Networking for Reverex DX. This then, in turn, reduced the number of questions that came my way regarding the networking tools and made my life so much easier during development as networking tasks that were always delegated to me could be more fairly delegated across the entire development team.

## Conclusions

It’s exceedingly difficult to write games, and especially difficult to write games with good and functioning Netcode. It’s not a task that can handled simply by just one person working alone, and toolsets are vital to making any project of scale work. We could’ve used a more complex and feature-rich API in retrospect since I had to replicate most of the capabilities of a more complex API anyway. However, it was still quite eye-opening to see all of the potential pitfalls that come with trying to build a networked game from such a low level.

While the original goal of having networking being a backburner idea was misguided, it was fun learning how to create such a complex system and get it mostly working. It was quite a lot of work to write the NetUtil package and if I had known how daunting it was from the beginning, I probably would’ve made a much more deliberate plan and only developed one internal framework and made the Syncronized Variables and NetRoutines just abstractions of that one framework.

However, a lot of what I wrote wasn’t because I wanted to write a comprehensive networking toolset, but rather, I wanted to reduce the herculean task of developing the entire Netcode for Reverex myself. Many of the first few implementations were made just to make my life easier, and I had just documented them and shipped them for others to use.

Just shipping scratch APIs with fancy documentation would never work to get people to use them. They had massive blindspots in the “how do others use this” category that I couldn’t fix until I started designing more thoroughly and discussing more proactively with the rest of the programming team. Ultimately, it’s more important to tailor the design of something toward other people on the project than just developing what seems good for me.

# Citations.

#### [1] “Steam Networking (Steamworks Documentation),” *partner.steamgames.com*. <https://partner.steamgames.com/doc/features/multiplayer/networking>

#### [2]‌ “linkachugaming0’s Profile - Member List - CurseForge,” *Curseforge.com*, 2025. <https://legacy.curseforge.com/members/linkachugaming0/projects> (accessed Jan. 02, 2025). [

#### [3] Unity, “Unity Documentation,” *docs.unity.com*. <https://docs.unity.com/>

#### [4] “Introduction to Unity Serialization | Serialization | 3.1.2,” *Unity3d.com*, 2022. <https://docs.unity3d.com/Packages/com.unity.serialization@3.1/manual/index.html> (accessed Jan. 02, 2025).

#### [5] Unity Technologies, “Unity - Scripting API: Coroutine,” *Unity3d.com*, 2022. <https://docs.unity3d.com/2022.3/Documentation/ScriptReference/Coroutine.html> (accessed Jan. 02, 2025).

#### [A] Colin12345678910, “GitHub - Colin12345678910/COSC3P99: Contains several code snippets from Reverex DX.,” *GitHub*, 2025. <https://github.com/Colin12345678910/COSC3P99> (accessed Jan. 03, 2025).

#### [A.1] Colin12345678910, “COSC3P99/Colin\_D/Whoami.cs at main · Colin12345678910/COSC3P99,” *GitHub*, 2025. <https://github.com/Colin12345678910/COSC3P99/blob/main/Colin_D/Whoami.cs> (accessed Jan. 03, 2025).

#### [A.2] Colin12345678910, “COSC3P99/Colin\_D/Syncronized.cs at main · Colin12345678910/COSC3P99,” *GitHub*, 2025. <https://github.com/Colin12345678910/COSC3P99/blob/main/Colin_D/Syncronized.cs> (accessed Jan. 03, 2025).

#### [A.3] Colin12345678910, “COSC3P99/Colin\_D/NetAction.cs at main · Colin12345678910/COSC3P99,” *GitHub*, 2025. <https://github.com/Colin12345678910/COSC3P99/blob/main/Colin_D/NetAction.cs> (accessed Jan. 03, 2025).

#### ‌[A.4] Colin12345678910, “COSC3P99/Colin\_D/NetRoutine.cs at main · Colin12345678910/COSC3P99,” *GitHub*, 2025. <https://github.com/Colin12345678910/COSC3P99/blob/main/Colin_D/NetRoutine.cs> (accessed Jan. 03, 2025).

#### 